Multi-threaded System for Activating a Process Using a Script Engine and Publishing Data Descriptive of the Status of the Process

REFERENCE TO APPENDIX A

TECHNICAL FIELD

In general, the invention relates to the retrieval of information descriptive of the status of a process, and more particularly, to network-based systems for accessing the information related to the status of any computer executable process executing either locally or remotely.

BACKGROUND OF THE INVENTION

Nowadays, almost all computing devices within the workplace are tied to a network. Many large and small companies alike employ a network to disseminate information throughout the organization. Typical uses of networks include linking multiple computers throughout the organization together, providing access to company wide information via a corporate intranet or the Internet, and connecting multiple computers to a specific printer or fax machine. The pervasive use of networks within the workplace can also be seen frequently within many households, where computers are commonly linked to a network, such as the Internet, to provide WEB access and e-mail messaging services.

In networked computing environments such as those described above, it is often desirable for a user to determine the status of a process that was initiated by their computer, such as the status of a document sent to a remote printer. According to many conventional computing systems, the status of the print job can be easily determined and even viewed with a simple network printer status tool that monitors the printing process as it is in action. Many other types of processes can also be initiated and monitored via a network in a well-known and conventional manner, as long as the tool used to communicate the status information is written specifically for that process. In the example of the print job, to communicate print status, the tool performing this action is programmed specifically for this purpose. The same print status tool cannot communicate the status of other tasks such as the delivery of an e-mail message or the detection of a computer virus. Thus, the client that invoked the print job status tool cannot use the same tool to view the status of a separate and independent process.

Because of this limitation, obtaining status information related to one or more independent processes requires a customized tool for each individual process. This requirement is undesirable, however, in situations where the overall process involves multiple independent sub-processes, such as in a distributed computing/networking environment.

The distributed compilation of a large software program provides a good example of how related, yet independently executed processes or tasks can occur within a distributed computing environment. In this example, several developers, often from different groups within an organization, share the responsibility of successfully compiling a software application. This procedure requires that each developer compile their developed source code, and then they must merge their code with that of the other developers to yield a fully operational end product. In order to successfully accomplish this, each developer must be able to monitor any one of the distributed compile processes at any time (i.e., another developer's compile process). However, due to the limitations imposed by conventional monitoring systems, there is no easy way for developers to monitor the status of each independent process in execution. The developer is only able to obtain the status information relative to his or her own compile process.

To overcome the limitations of conventional status information retrieval systems, a convenient means of determining the status of one or more independently executing processes is required. Also, a convenient means of retrieving status information related to the executable processes from any location is needed.

SUMMARY OF THE INVENTION

The present invention fulfills these requirements by employing a multi-threaded process management system that maintains consistent status information related to an executable process, and facilitates the exchange of the information between machines over a network. The information is collected by the process management system throughout the execution of the process and stored in a retrievable data structure. Any machine having access to the network can communicate with the process management system, and subsequently retrieve the data structure containing the status information.

Another feature of the invention allows a local or remote user to invoke a process by communicating with the process management system from over a network. The process management system invokes the requested action on the user's behalf by activating a script engine that interprets a sequence of executable script commands and/or routines that initiate and enable the process. The status information related to the invoked action is subsequently stored in a publicly accessible data structure as the process is in execution. Because the method of invoking the process and retrieving the process information is not dependent upon the process itself, the invention can be used in connection with any computer executable process.

Additional features and advantages of the invention will be made apparent from the following detailed description of illustrative embodiments that proceeds with reference to the accompanying figures.

BRIEF DESCRIPTION OF THE DRAWINGS

While the appended claims set forth the features of the present invention with particularity, the invention, together with its objects and advantages, may be best understood from the following detailed description taken in conjunction with the accompanying drawings of which:

Figure 1 is a block diagram generally illustrating an exemplary computer system on which the present invention resides;

Figure 2 is a high-level block diagram illustrating the overall architecture of the invention;

Figure 3 is a simple flow chart illustrating the interaction between the user interface and the process management system to invoke an executable process;

Figure 4 is a flow chart illustrating the interaction between the user interface and the process management system to obtain status information related to an executable process distributed amongst a plurality of nodes over the network;

Figure 5 is a block diagram illustrating the basic components of the process management system and the interactivity between the various components;

Figure 6 is a flow chart illustrating the overall process of communication between the user interface and the process management system.

DETAILED DESCRIPTION OF THE INVENTION

Turning to the drawings, wherein like reference numerals refer to like elements, the invention is described hereinafter in the context of a computing environment. Although it is not required for practicing the invention, the invention is described as it is implemented by computer-executable instructions, such as program modules, that are executed by a PC (PC). Generally, program modules include routines, programs, objects, components, data structures and the like that perform particular tasks or implement particular abstract data types. In the illustrated embodiment described in detail herein, the various program modules or routines employed by the invention will be described according to the Microsoft Component Object Model (COM) and Distributed Component Object Model (DCOM). As is well known in the art, COM is an object oriented specification and software development methodology that allows independent program modules, or components, to connect and interact with relative ease. Likewise, DCOM extends the capabilities of COM by providing remote processing capability to COM object. However, while the invention will be described in the context of COM and DCOM embodiment, the invention is not meant to be limited to any specific architecture.

The invention may be implemented according to other methodologies or protocols such as COBRA (Common Object Request Broker Architecture), SOM/DSOM (Distributed System Object Model), or the Java-based Remote Method Invocation (RMI) protocol. Moreover, languages/methodologies based on modular or object-oriented programming such as C/C++, or those requiring procedural execution can also be used.

Although the invention is described hereinafter in the context of a PC-based embodiment, other hardware environments may also make use of the invention. For example, the invention may be practiced in hand-held devices, multi-processor systems, microprocessor-based or programmable consumer electronics, network PCs, minicomputers, mainframe computers and the like. The invention is also well suited for use in distributed computing environments, where tasks are performed by remote processing devices that are linked through a communications network. In a distributed computing environment, program modules may be located in both local and remote memory storage devices.

While the invention may be incorporated into many types of computing environments as suggested above, the following detailed description of the invention is set forth in the context of an exemplary general-purpose computing device in the form of a conventional PC 20.

< The Exemplary Computing Environment Computing Environment >

Before describing the invention in detail, the computing environment in which the invention operates is described hereinafter in connection with Figure 1.

The PC 20 includes a processing unit 21, a system memory 22, and a system bus 23 that couples various system components including the system memory to the processing unit 21. The system bus 23 may be any of several types of bus structures including a memory bus or memory controller, a peripheral bus, and a local bus using any of a variety of bus architectures. The system memory includes read only memory (ROM) 24 and random access memory (RAM) 25. A basic input/output system (BIOS) 26, containing the basic routines that help to transfer information between elements within the PC 20, such as during start-up, is stored in ROM 24. The PC 20 further includes a hard disk drive 27 for reading from and writing to a hard disk 60, a magnetic disk drive 28 for reading from or writing to a removable magnetic disk 29, and an optical disk drive 30 for reading from or writing to a removable optical disk 31 such as a CD ROM or other optical media.

The hard disk drive 27, magnetic disk drive 28, and optical disk drive 30 are connected to the system bus 23 by a hard disk drive interface 32, a magnetic disk drive interface 33, and an optical disk drive interface 34, respectively. The drives and their associated computer-readable media provide nonvolatile storage of computer readable instructions, data structures, program modules and other data for the PC 20. Although the exemplary environment described herein employs a hard disk 60, a removable magnetic disk 29, and a removable optical disk 31, it will be appreciated by those skilled in the art that other types of computer readable media which can store data that is accessible by a computer, such as magnetic cassettes, flash memory cards, digital video disks, Bernoulli cartridges, random access memories, read only memories, and the like may also be used in the exemplary operating environment.

A number of program modules may be stored on the hard disk 60, magnetic disk 29, optical disk 31, ROM 24 or RAM 25, including an operating system 35, one or more applications programs 36, other program modules 37, and program data 38. A user may enter commands and information into the PC 20 through input devices such as a keyboard 40 and a pointing device 42. Other input devices (not shown) may include a microphone, joystick, game pad, satellite dish, scanner, or the like. These and other input devices are often connected to the processing unit 21 through a serial port interface 46 that is coupled to the system bus, but may be connected by other interfaces, such as a parallel port, game port or a universal serial bus (USB). A monitor 47 or other type of display device is also connected to the system bus 23 via an interface, such as a video adapter 48. In addition to the monitor, PCs typically include other peripheral output devices, not shown, such as speakers and printers.

The PC 20 may operate in a networked environment using logical connections to one or more remote computers, such as a remote computer 49. The remote computer 49 may be another PC, a server, a router, a network PC, a peer device or other common network node, and typically includes many or all of the elements described above relative to the PC 20, although only a memory storage device 50 has been illustrated in Fig. 1. The logical connections depicted in Fig. 1 include a local area network (LAN) 51 and a wide area network (WAN) 52. Such networking environments are commonplace in offices, enterprise-wide computer networks, intranets and the Internet. As will be discussed later on in greater detail with regards to the invention, the remote computer 49 is a primary node on which a process management system executes.

When used in a LAN networking environment, the PC 20 is connected to the local network 51 through a network interface or adapter 53. When used in a WAN networking environment, the PC 20 typically includes a modem 54 or other means for establishing communications over the WAN 52. The modem 54, which may be internal or external, is connected to the system bus 23 via the serial port interface 46. In a networked environment, program modules depicted relative to the PC 20, or portions thereof, may be stored in the remote memory storage device. It will be appreciated that the network connections shown are exemplary and other means of establishing a communications link between the computers may be used.

In the description that follows, the invention will be described with reference to acts and symbolic representations of operations that are performed by one or more computers, unless indicated otherwise. As such, it will be understood that such acts and operations, which are at times referred to as being computer-executed, include the manipulation by the processing unit of the computer of electrical signals representing data in a structured form. This manipulation transforms the data or maintains it at locations in the memory system of the computer, which reconfigures or otherwise alters the operation of the computer in a manner well understood by those skilled in the art. The data structures where data is maintained are physical locations of the memory that have particular properties defined by the format of the data. However, while the invention is being described in the foregoing context, it is not meant to be limiting as those of skill in the art will appreciate that various of the acts and operations described hereinafter may also be implemented in hardware.

The present invention accesses status information that is generated by one or more nodes as a result of the execution of a computer-executable process. As described herein, the executable process is any computer-executable code, routines or sequence of instructions that enable a specific task or action to be invoked by a computer. This includes simple tasks, such as the transferring of a file, or more complex tasks such as performing a complete build of a software product or controlling an electro-mechanical process. According to the methods of the invention, the process is implemented according to a standard scripting language, such as those defined by the ECMAScript standard. This includes script languages such as Microsoft JScript and VBScript, or Netscape/AOL JavaScript. The script language, or code, is then parsed and executed by a script engine, which is a script language interpreter that resides on the node responsible for executing the process. Also, the status information related to the executable process can be any data or parameters that are generated by the script code that enables the process. This includes information such as the current state of the process, the characteristics of the computer system in which the process is being run, data objects, arrays and any other data that can be put into a script variable. Because the process is implemented as executable script code, the status information that is generated is dependent upon the customized design of the script itself.

Turning to Figures 2 and 3, a user interface 100, executing on a personal computer such as that shown in Figure 1, is capable of communicating with a remote computer 49 over an established network connection 51/52. The user interface 100 is best characterized as the portion of a computer application that displays useful information to the user. For instance, many Windows-based applications employ a graphical user interface having various menus, display objects and buttons that represent specific features of the application. In general, the user interface 100 contains text, graphical objects and/or instructions that allow the user of the computer 20 to perform various tasks. Virtually any computer-implemented process or application that renders the display of text or images to a monitor 47 for performing a task is a user interface, and falls within the scope of the invention.

In the illustrated embodiment, the user interface 100 is a web page. The web page is generated by a web browser application 102, which is capable of processing web-based content (including script code) received from over a network 51/52, and displaying text and graphical information to the user from the monitor 47. Embedded within the web page 100 is an Active X object, referred to as the RemoteMTScriptProxy object 104(a), that employs Active Scripting technology. Active Scripting is a well-known technology that allows a script engine to be encapsulated into a COM object, such as an Active X object. The RemoteMTScriptProxy object 104(a) enables script code contained within the web page 100 to communicate with other machines or objects remotely. In particular, the script code is written to perform remote processing tasks. One such task is to establish a connection with a remotely located primary node 106, on which a process management system 108 resides. Another task is to invoke the execution or action of a process that is maintained by the process management system 108. As discussed previously, the process is written in executable script code. In order to activate this code, which resides within a physical memory location on the primary node 49, the user interface 100 must submit a request to the process management system 108 (event 204). The user interface accomplishes this by accepting a command to activate the process as input (event 200). The command is input into the user interface screen manually by a user of the Web browser 102, such as by clicking an action button embedded within the user interface 100 with the mouse 42. Clicking the action button results in the initiation of the script code contained within the web page 100, which in turn activates the

RemoteMTScriptProxy object 104(a) to communicate with the primary node 106. While not shown in Figures 2 or 3, the command can also be entered from a command line entry field that implements the connectivity of the RemoteMTScriptProxy object 104(a). Or, as another alternative, the command can be input without human intervention by a separate application or process that called the user interface. Any means by which the desired process or action to be invoked is specified to the user interface screen is acceptable. The input command also includes descriptive parameters that are necessary to properly execute the process. For instance, a command to execute a process for transferring a file from one location to another must include other descriptive parameters/information such as the name of the file to be transferred, its current location, and the desired location.

The other task of the user interface is to establish a COM/DCOM connection with a process management system 108 executing on a primary node 106. The process management system 108 performs numerous tasks, which include storing status information related to the executable process, managing multiple script threads, processes, and script engines 114/116, and maintaining connections with one or more user interfaces 100 or remote machines 122. The full architecture and operation of the process management system is discussed in greater detail hereinafter. The process management system provides a means by which any computer executable process can be implemented, accessed, and maintained within a distributed or non-distributed processing environment.

With reference again to Figure 2, once the user command is received by the user interface 100, it is processed, and a connection with the process management system 108 is established by activating the script code contained within the web page 100 (event 202). As is commonly known in the art, COM/DCOM allows remote and local applications to interact with each other and the operating system upon which they execute through collections of function calls known as interfaces. The established DCOM connection implements the IConnectedMachine interface 110, which allows the user of the web page 100 to communicate with and access information directly from the process management system 108.

Once the IConnectedMachine interface 110 is created, the command received by the user interface 100 is transmitted to the process management system 108 (event 204). As shown in Figure 2, the process management system implements a RemoteMTScriptStub object 104(b). This is an instantiated version of the RemoteMTScriptProxy object 104(a), and enables the remote communication abilities of DCOM. The command is received by the RemoteMTScriptStub 104(b), which in turn activates the DLocalMTScriptEvents interface 112. This interface activates one or more script engines 114/116 to execute the script code contained within memory 50 that corresponds to the command—the desired process (event 206).

As the process is in execution, the script code communicates with the process management system via the IGlobalMTScript interface 118 (event 210). This interface allows the scripts in execution by the process management system 108 to send status information about the process to the process management system. The status information is then stored within a data structure (not shown) that is maintained by the process management system (event 212). The IGlobalMTScript interface 118 is also used by the scripts to notify the process management system 108 of events that occur during process execution, such as the occurrence of an error or the completion of the process. To further facilitate the exchange of information, any remote machines 122 connected to the process management system through the IConnectedMachine interface 110 will send event notifications via the DRemoteMTScriptEvents interface 120. This is the primary interface that the connected machines 122 use to communicate with the process management system.

A critical feature of the invention is the ability to allow one or more nodes to access the status information generated by the scripts in execution from over the network. Any machine having the appropriate access to the network upon which the primary node 49 is located, can access the process management system, and thus access the status information. Appropriate access refers to the ability of the machine to penetrate a network security infrastructure that may be in place within the network of the primary node 49. Machines that are capable of connecting with the process management system 108 can poll it for data using a standard polling mechanism, or can explicitly request the data from a user interface 100; and subsequently receive the data structure containing the status information. Figure 2 and the flowchart of Figure 4 further illustrate the interaction that takes place between the process management system 108, and any connected nodes 122 and 124 to provide this ability.

The machine boundaries 130 as shown in Figure 2, indicate the locality of the various nodes that are capable of connecting with the process management system through the IConnectedMachine interface 110. In one instance, one or more user/client nodes 124 can connect with the process management system through a user interface 100 such as a web page. This method of accessing the process management system was discussed in foregoing paragraphs of the description. In another instance, one or more remote nodes 122 can connect to the process management system 108 on the primary node 49. This method of accessing the process management system does not require a connection being established from that user interface 100. Instead, the script code within memory 50 on the primary node is capable of formulating connections with other process management systems executing on one or more remote nodes 122. The following paragraphs further describe this ability.

Script code can be written to connect directly to COM/DCOM objects. Because of this ability, the script code that is executed by the process management system 108 on the primary node 49 can in turn connect to other process management systems executing on the one or more remote nodes 122 by formulating a DCOM connection, and implementing the IConnectedMachine interface 110. The remote nodes have the same general internal architecture as that of the primary node 49. Each has a process management system 108, a local memory for accessing executable script code, and one or more script engines for processing and interpreting executable scripts. However, each node contains a different set of script code that performs a different task than that of the primary node, and contains different script engines for executing this code. Thus, the executable process can be executed solely by the primary node, as illustrated in Figure 3, or can be distributed amongst a plurality of independent nodes over the network as illustrated in Figure 4.

With reference to Figure 4, the user/client nodes 124 that connect through the user interface 100 obtain status information related to the executable process by polling the process management system 108 on the primary node 49 (event 300). Once this request for information is received by the process management system through the IConnectedMachine interface 110, the process management system accesses the data structure that it maintains during the execution of the process (event 302). As discussed previously, the script code in execution on the primary node updates this data structure consistently. Yet, when the process is distributed amongst a plurality of nodes, each of the nodes store status information generated by the scripts into its own data structure. Therefore, in order to return complete process information to the user, the process management system 108 on the primary node must be able to obtain status information from all of the nodes 122 that execute the overall process.

To accomplish this, the process management system 108 consistently polls the one or more remote nodes 122 for updated information over the IConnectedMachine interface 110 (event 304). Upon receiving the request, the one or more remote nodes pass the information stored within their respective data structures to the primary node. This information is then stored into the data structure on the primary node, and sent to the requesting user/client node 124 (events 306 -310). Because the scripts are able to formulate connections with other process management systems, data can be easily exchanged between a plurality of nodes within a distributed network. Each machine communicates with the next via the IConnectedMachine interface 110, and event notifications are fired to each machine using the DRemoteMTScriptEvents interface 120. Furthermore, the connections enable the plurality of nodes to submit commands and/or requests to invoke the action of the executable process, thus increasing the ability of the nodes to work together to perform complex tasks.

The flowchart of Figure 6 summarizes the steps illustrated in Figures 3 and 4, and shows the overall communication flow of the invention. Specifically, it shows the logical steps that are taken by the process management system and a user/client 124 or remote node 122, to facilitate the exchange of status information. Moreover, it illustrates the sequence of steps that are taken by the user/client nodes 124 and/or remote nodes 122 to connect to the process management system and invoke the executable process. As shown, the process management system 108 responds to various requests and commands received from the user, and responds accordingly. Certain events are handled directly by the script code that implements the process, while other events are handled by the process management system itself.

To promote a better understanding of the invention, a detailed description of the internal architecture of the process management system is described below. While the general description is in the context of specific class names, objects, and function calls implemented using COM/DCOM, the invention is not meant to be limited to the particular implementation disclosed herein. Those skilled in the art will recognize that the invention can be implemented according to various programming techniques and methodologies. The process management system will be discussed with reference to the block diagram of Figure 5.

The process management system is a computer executable application that resides upon a primary node 49, and is capable of executing multiple script threads. Scripting threads are simply threads that execute according to a scripting language, such as JavaScript, VBScript or PERL. Programmers often prefer to use scripting languages rather than compiled languages because scripting languages provide an easy means of executing source code that avoids many of the complexities of the operating system. For instance, scripts are interpreted and processed directly by a separate program called a script engine 114/116, and so do not need to rely explicitly on the message passing facilities of the operating system in order to execute, as in some machine executable languages

One of the primary purposes of the process management system 108 is to provide a platform upon which any computer executable process can operate. To achieve this, the process to be executed cannot rely upon the process management system to interpret, compile or process the source code that defines the process. With this requirement in mind, a customized set of script code that is written to perform a specific task or procedure, is used to implement the process. The process management system 108 is only responsible for providing the means of activating the script code, monitoring it during its execution, and sharing status information generated by the script code. Thus, the process management system is not designed specifically for the particular executable process, but is instead a framework upon which any process can be executed. It does not execute the process (interpret the source code) directly, but simply hosts one or more script engines for interpreting executable scripts. Furthermore, the process management system does not require any process specific mechanisms to collect and retrieve status information. It simply implements interfaces that allow data to be exchanged without regard to the methods or functions of the source code that implements the process. For these reasons, the invention is suitable for executing any process.

The process management system 108 is comprised of various inter-dependent components for performing various tasks. Because one of the primary functions of the process management system is to provide multi-threaded scripting behavior, a base thread-aware class provides the basic building blocks for the components of the process management system. This base class, named *CThreadComm*, provides all the basic behaviors of a script thread, including creation, initialization, cross-thread communication (both synchronous and asynchronous), and thread locking for cross-thread data access. Derivative classes of *CThreadComm* include, but are not limited to, *CMTScript* 406, *CMachine 420, CProcessThread* 424 and *CScriptHost* 408. Each of the classes implements a different thread, or object, of execution within the process management system. This offers a distinct advantage over other scripting technologies, such as Windows Script Host, which does not support multi-threaded execution.

Threads communicate to each other using mechanisms provided by *CThreadComm*. These mechanisms are similar to the way the Windows operating system handles window messages. For instance, *CThreadComm* implements a SendToThread() method that sends a synchronous message and waits for a response. If asynchronous notification is desired, then an object can use PostToThread(). Both objects participating in the communication must derive from *CThreadComm.* Also, to further support multi-threaded operation within the process management system, each of the classes of *CThreadComm* provides different levels of functionality and threaded behavior. The definitions and coded based implementations of these classes are shown in Appendix A.

In Figure 5, the various components that comprise the process management system are shown. The primary components are the script hosting module 400, the process execution module 402, the remote communication module 404, and the CMTScript module 406. A description of each of the components follows.

Script Hosting Module

The script hosting module 400 is the part of the process management system that manages the script engines 114 and multiple threads of script execution. It also provides behavior to the script engines that gives them extended capability. Some of this additional behavior includes the ability for the scripts in different threads to communicate, spawn new script threads, and other extended tasks beyond the normal capabilities of the script engine. The script hosting module contains three major components: 1) source code designed for managing and hosting one or more Active Script engines, 2) implementation of the IGlobalMTScript interface 118, and 3) code that fires events into the appropriate script engines using the DLocalMTScriptEvents interface 112.

The process management system integrates with the script engine 114 via the *CScriptHost* class 408. When this class is called, due to a request to execute script code, it takes a path to the script file that corresponds to the request. The process management system then instantiates the appropriate script engine 114 to enable the process. One of the requirements of Active Script engines is that they be provided a script site, which is simply an object on which function callbacks can be made. *CScriptHost* creates a *CScriptSite* class 408 and passes this object to the script engine. The process management system then loads the script file that corresponds to the desired process to be executed, and passes it to the script engine. The script engine parses the script file, executes global code, and returns. In most cases very little is done at this stage besides initializing global variables.

The actual execution of meaningful code is not performed until the ScriptMain() event (an event defined by the process management system, not Active Scripting) is fired. This is because of the way in which the script engine operates. By convention, the script engine cannot be transitioned to the CONNECT (ready to execute) state until it has returned from the initial parsing of the script, and the engine must be in the CONNECT state for event firing to work. At this point the script engine is transitioned to the CONNECT state, which hooks up the *DLocalMTScriptEvents* interface (described below). Then the ScriptMain() event is fired, which is when the script performs all of its behavior. Once ScriptMain() returns, the script is considered finished and is shut down and deleted.

The process management system has what is called a "primary" script (not shown). It is the only script for which the user tells the process management system its location. The primary script is automatically loaded by the process management system when it is initiated, and if a failure happens finding, loading, or parsing this script then the process management system reports the error and terminates. All other scripts are spawned by the primary script using methods on IGlobalMTScript 118. Thus, requests to activate the executable process and other events are fired into the primary script first. The primary script then activates the specified script engine through the DLocalMTScriptEvents interface 112 to begin process execution. Once this script is fully executed, ScriptMain() is returned to signal the termination of the process, and the script is deleted. However, unlike other scripts, the primary script does not terminate when it returns from ScriptMain().

If the primary script terminates (with an error, for example), then the process management system cannot continue and subsequently terminates. Script errors are reported by the script engine 114 to the process management system 108 via the script site 410. Once a script error occurs, then that script can no longer execute and must be terminated. The process management system 108 gives the script a chance to clean things up and log the error by firing another method implemented by the process management system, OnScriptError(), before terminating the script. The script can choose whether the error should be reported to the user.

As discussed previously, the script hosting module 400 provides behavior to the script engines that gives them extended capability. While Active Script engines 114 allow users to write and execute script logic, more complex behavior such as launching other processes or threads is not provided by the script engine itself. For Active Scripting, this type of behavior must be provided by the process management system or by creating COM objects that can be used directly by the script engine. In the present invention, this extended behavior is provided by the process management system according to a global object property (not shown). Specifically, when an Active Script engine 114 encounters an identifier within the script code, such as a function name or variable that it cannot resolve, it will check with the global object to see if it can provide that function or variable. In this way, executable tasks or functions that are not generally capable of being performed by the script engine 114 can be performed. By using the global object, the process management system 108 is able to allow scripts to spawn other scripts, send email, synchronize with each other, launch processes, etc. The software developer can customize the functions or variables provided by the global object to fit their specific requirements. Also, the global object implements the IGlobalMTScript interface 118 to allow the process management system 108 and the script engine 114 to interact.

Process Execution Module

The process execution module 402 is the part of the script host that handles the execution and monitoring of local processes on the machine. Local processes can be any executable applications, functions, or the like that reside within the application program space 36 or a memory location 50 on the primary node 49. Any of the scripts can launch a local process by calling the global object. The script specifies whether the process should be executed synchronously or asynchronously, whether its window should be minimized or not, whether to capture its output, etc. The process execution module spawns the local process, monitors it, and reports status information back to the script. This status information is kept around for some time after the process terminates. This allows the script to have access to the captured output and exit code of the process even after it has terminated. Once a dead process is purged, then attempts to obtain the exit code will fail with an invalid parameter error.

Remote Communication Module

Along with allowing multi-threaded scripts and rich execution of local processes, the third main goal of the process management system is to make it easy to obtain and share status information with other machines within a distributed network environment. To accomplish this, the process management system 108 registers DCOM objects on the machine that can be instantiated (communicated with) on other machines.

When the process management system is run, it registers the RemoteMTScriptStub 104(b) as a remotable object. This object implements *IConnectedMachine* 110 and has *DRemoteMTScriptEvents* 120 as an event interface. As a remotable object, other machines can instantiate the object and communicate to it over the network. As is commonly known in the art, DCOM allows the object to be communicated with as if it were on the primary node 49 even when it is on a remote node 122/124 (both local and remote communication are possible). Once the object is called, the remote node has full access to the IConnectedMachine 110 and DRemoteMTScriptEvents 120 communication interfaces. If the object is called from a user interface such as the HTML-based interface of Figure 2, the RemoteMTScriptProxy object 104(a) must be used to facilitate the communication interfaces. Those skilled in the art will recognize however, that other types of user interfaces can be used which do not require the usage of the RemoteMTScriptProxy object 104(a/b). For example, a Win32 based user interface can be programmatically designed to facilitate the remote communication without the need for a proxy object. Nonetheless, the user interface 100 as shown in the illustrated embodiment is capable of processing Web-based content.

IConnectedMachine 110 is the interface implemented by the Remote Communication Module 404 that allows remote nodes 122/124 to access the process management system. Through this interface, the remote machine can perform three tasks, which are 1) obtain information about the machine the process management system

108 is running on, 2) submit a request to activate an executable process to the primary script, and 3) retrieve the data structure that contains status information related to the executable process. The information about the process management system 108 includes the OS, platform, version, and any other system information that the connected machine may be interested in. As for the second task, a request to activate an executable process is submitted by the remote nodes 122/124 by calling the IConnectedMachine interface 110, and passing the command to invoke the action to the primary script. The primary script then invokes the scripts and script engines 114 that correspond to the request. It is up to the scripts to determine how to interpret the command and take the appropriate action to fulfill the request. With regards to the third task, the remote nodes 122/124 retrieve the data structure by accessing the process management system 108 through IConnectedMachine 110. As will be discussed later, the data structure that is sent can be a PublicData structure that contains complete status information, or a StatusValue data structure that contains information regarding a particular event.

As scripts are in execution, they can send information to any remote nodes 122/124 that are connected to the process management system 108 via IConnectedMachine. For example, the script can send a notification to a connected node to alert it that the data structure has changed, or that the executable process is complete. This can be done using the event interface DRemoteMTScriptEvents (not shown in Figure 5 but in Figure 2). The one or more remote nodes 122/124 wishing to receive notification of certain events give the process management system an event sink for DRemoteMTScriptEvents 120. Then, the script in execution can call the global object to send a message to any connected machines. If no machines are connected, the message is discarded.

CMTScript Module

The CMTScript module 406 is the main glue that connects all of the other components of the process management system 108 together. It acts as a liaison between the remote communication module 404 and the script hosting module 400 so that they can send messages to each other. Requests received over the IConnectedMachine interface 110 from the one or more connected nodes 122/124 or from local processes 418 are passed from the remote communication module 404 to the script hosting module 400 using the methods of CMTScript. The script hosting module also delegates certain tasks to the CMTScript module 406 such as the launching of the primary script, the running of processes, and the storage of data that is generated by the scripts. The CMTScript module also keeps track of all current and recently executed scripts and processes that are spawned during process execution. Furthermore, the CMTScript module handles command-line parameters and the program registry, and stores/loads the default configuration for the process management system 108 (*.exe).

During process execution, the scripts generate status information related to the executable process. To ensure that this information is shared between nodes, the process management system 108 maintains a data structure that is accessible to the one or more nodes 122/124 connected with the process management system 108. This data structure is called PublicData 416, and is implemented by the CMTScript module 406. Any data that a script puts into PublicData 416 is immediately accessible from a remote machine

122/124. This feature essentially allows the scripts to "publish" information about the task they are performing, or store data that they have obtained from some other source, etc. This can be data such as the current state of the process (idle, busy, completed) or the type of process in execution (master, slave, standalone). Any data whatsoever can be put into the PublicData structure; it is up to the scripts to determine what gets placed there. PublicData is the data structure that is returned to the user interface 100 or the remote nodes 122 when the process management system is polled for status information.

The CMTScript module 406 also provides two other data structures to facilitate the exchange of information between nodes. These are the StatusValue 412 and PrivateData 414 data structures. StatusValue 412 is similar to the PublicData structure in that it is visible to the one or more connected nodes 122/124 via the IConnectedMachine interface 110. However, the data is limited to only a maximum number of integer values *n*, where *n* is variable (by a software developer) according to the needs of the process management system. The advantage of the StatusValue 412 data structure is that it stores less information, and so can be retrieved by remote machines 122/124 much faster than the larger PublicData 416. For example, a script may want to simply provide an indication as to whether or not an error has occurred in its processing. Rather than send the complete set of status information to the user/client 124 via PublicData 416, the StatusValue data structure 412 can be sent/retrieved to indicate if an error happened (or perhaps show a count of errors). StatusValue provides a quick indication of an event that occurs during process execution, and can be quickly and efficiently retrieved by remote nodes.

Any script thread can read from or write to either of these data structures mentioned above, and all other script threads will be able to see the changes. Thus, PublicData and Private data are accessible remotely by one or more nodes, and locally by script threads in execution on the machine. The PrivateData data structure 414 differs from PublicData 416 and StatusValue 412, however, in that it is not accessible to the remote machines 122/124 via the IConnectedMachine interface 110. Instead, PrivateData is accessible to only the script threads that are spawned during process execution. PrivateData can be used by the script threads to store information that relates to the execution of the process, such as temporary information that was retrieved from other machines, data regarding what commands have been sent from a remote machine 122/124, or a list of remote machines 122/124 that are currently connected. The PrivateData data structure is useful for ensuring cross-thread communication between multiple script threads, particularly when the process is distributed amongst a plurality of nodes.

Overall, the present invention provides a means in which computer executable processes can be invoked and subsequently monitored by one or more nodes from over a network. The process is written in a scripting language that is executable by a script engine, and can be executed by a single primary node or distributed amongst a plurality of nodes tied to the network. By distributing the process amongst a plurality of nodes, the process is broken down further into a plurality of sub processes, where each sub process is independently executed by one of the plurality of nodes. This is desired in many situations to decrease the amount of time required to complete a full computer executable process by a single machine. Distribution of the process across multiple nodes also avoids the drainage of resources that occurs when a single machine is left to perform large scale computing tasks. The present invention provides an easy means by which the processes can be run independently, but monitored and executed dynamically from over the network.

Unlike conventional process monitoring systems, the process management system 108 operates independently of the executable process. In this respect, it operates regardless of the specific mechanisms, OS dependencies or methods of the programming language in which the process is implemented. Instead, it simply manages one or more script engines, which provide the ability to interpret and compile a process written in script. Another aspect of the invention is that the process management system 108 provides the necessary communication channels needed to exchange data generated by the scripts between nodes, interact with the scripts to store status information related to the executable process, and send notification of events to any connected machines.

Still further, the present invention manages multiple scripts threads during the execution of the process, and provides mechanisms by which these script threads may communicate with each other, and other processes. These mechanisms are provided by a global object, which is an object that extends the capabilities of the scripting engines. This is necessary because some script engines (e.g. JScript) do not have any built in mechanisms to allow scripts to perform events that are extrinsic to the operation of the script code. Examples include the ability to call local processes, spawn new script threads, etc. Ultimately, the invention can be practiced in any situation where process execution and status information retrieval is to be accomplished from over a network.

All of the references cited herein, including patents, patent applications, and publications, are hereby incorporated in their entireties by reference.

In view of the many possible embodiments to which the principles of this invention may be applied, it should be recognized that the embodiment described herein with respect to the drawing figures is meant to be illustrative only and should not be taken as limiting the scope of invention. For example, those skilled in the art will recognize that the various components and mechanisms of the process management system as detailed in the figures can be implemented according to other programming methodologies, including but not limited to DSOM, COBRA, and RMI. Furthermore, those of skill in the art will recognize that the elements of the illustrated embodiment shown in software may be implemented in hardware and vice versa or that the illustrated embodiment can be modified in arrangement and detail without departing from the spirit of the invention. Therefore, the invention as described herein contemplates all such embodiments as may come within the scope of the following claims and equivalents thereof.

APPENDIX A:

Primary Class Implementations for the Multi-threaded Process Management System

```
//+--------------------------------------------------------------------------
//
//  Microsoft Windows
//  Copyright (C) Microsoft Corporation, 2000 - 2001
//
//  File:       classes.h
//
//  Contents:   Contains the class definitions for the primary classes used
//              by the MTScript host.
//
//----------------------------------------------------------------------------

//+---------------------------------------------------------------------------
//
//  Class:      CThreadComm (ctc)
//
//  Purpose:    Base class which handles cross-thread communication. This
//              is the only class with methods that can safely be called
//              by any thread. All other classes must have their methods
//              called by their owning thread (except the ctor/dtor and Init
//              which are called by the creating thread).
//
//----------------------------------------------------------------------------

class CThreadComm : public IUnknown, public CThreadLock
{
public:
    DECLARE_MEMCLEAR_NEW_DELETE();

    // Thread-unsafe member functions - can only be called by owning (or
    // creating) thread.

    CThreadComm();
   ~CThreadComm();

    HRESULT     StartThread(void * pvParams);
    BOOL        Shutdown(CThreadComm * pTarget);
    static DWORD  TempThreadRoutine(LPVOID pvParam);
    virtual DWORD ThreadMain() = 0;
    // Used to tell StartThread that the new thread is going and has picked
    // up all of its parameters.
    // If fSuccess is false, then the thread will be terminating
    // immediatly, and StartThread() will wait for it to do so.
    void          ThreadStarted(HRESULT hrThread) {
                                  _hrThread = hrThread;
                                  SetEvent(_hThreadReady); // Signal the other thread
                                  ::Sleep(100);          // Yield to the other thread
                                 }

    BOOL GetNextMsg(THREADMSG *tm, void * pData, DWORD *cbData);
    void Reply(DWORD dwReply);

    void SetName(LPCSTR szName);

    // ----------------------------------------------------------------
    //
    // Thread-safe member functions - can be called by any thread.

    HANDLE hThread() { return _hThread; }

    void  PostToThread(CThreadComm *pTarget, THREADMSG tm, void * pData = NULL, DWORD cbData = 0);
    DWORD SendToThread(CThreadComm *pTarget, THREADMSG tm, void * pData = NULL, DWORD cbData = 0);

    // End of thread-safe member list
    //
    // ----------------------------------------------------------------

protected:
    virtual BOOL Init();

    //
    // Every method on objects that derive from CThreadComm should have
    // VERIFY_THREAD at the beginning. VERIFY_THREAD ensures that the proper
    // thread is calling the method (i.e. it ensure that proper apartment rules
    // are being followed.)
    //
    inline void VERIFY_THREAD() { Assert(GetCurrentThreadId() == _dwThreadId); }

    HANDLE          _hThreadReady; // Event signaled when the new thread has set its return value
    HANDLE          _hCommEvent; // Event signaled when there is a message
    DWORD           _dwThreadId;
    HANDLE          _hThread;
    void *          _pvParams;
    HRESULT         _hrThread;   // The new thread sets this for initial success/failure.
private:

    //
    // MSGDATABUFSIZE: Max size of a thread message. The -40 is to keep the
    // MESSAGEDATA struct below 1K.
    //
    #define MSGDATABUFSIZE  (1024 - 40)

    //
    // MESSAGEDATA: All the data associated with a thread message.
    //
    struct MESSAGEDATA
    {
        MESSAGEDATA *pNext;
        THREADMSG    tmMessage;
        DWORD        dwResult;
        HANDLE       hResultEvt;
        DWORD        cbData;
        BYTE         bData[MSGDATABUFSIZE];
    };

  #if DBG == 1
    typedef struct tagTHREADNAME_INFO
    {
            DWORD    dwType; // == 0x1000
            LPCSTR   szName;
            DWORD    dwThreadID;
            DWORD    dwFlags;
    } THREADNAME_INFO;
```

```
#endif

    DWORD SendHelper(THREADMSG tm,
                     void *    pData,
                     DWORD     cbData,
                     BOOL      fSend,
                     HANDLE    hResultEvt);

    // ***************************
    //   THREAD-SAFE MEMBER DATA
    //   All access to the following members must be protected by LOCK_LOCALS()
    //
    MESSAGEDATA      *_pMsgData;   // Linked list of messages
    BOOL              _fInSend;    // True if we're inside SendToThread -
                                   //   needed to catch deadlock situations.

    MESSAGEDATA     * _pMsgReply;  // Place where we need to put our reply
    HANDLE            _hResultEvt; // Event used to indicate that the result
                                   //   is ready.
};


//+---------------------------------------------------------------------------
//
//  Class:      CMTScript (cmt)
//
//  Purpose:    Class which runs the main thread for the process.
//
//----------------------------------------------------------------------------

#define MAX_STATUS_VALUES 16 // Maximum allowed StatusValue values.
class CMTScript : public CThreadComm
{
    friend int PASCAL WinMain(HINSTANCE hInstance,
                              HINSTANCE hPrevInstance,
                              LPSTR     lpCmdLine,
                              int       nCmdShow);
    friend LRESULT CALLBACK
           MainWndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam);

    friend class CConfig;

public:
    CMTScript();
   ~CMTScript();

    DECLARE_STANDARD_IUNKNOWN(CMTScript);

    // Script Debugging helpers
    IProcessDebugManager * _pPDM;
    IDebugApplication    * _pDA;
    DWORD                  _dwAppCookie;

    CScriptHost *GetPrimaryScript();
    CProcessThread *GetProcess(int index);
    BOOL GetScriptNames(TCHAR *pchBuffer, long *pcBuffer);

    // These methods are thread safe.

    HRESULT             AddProcess(CProcessThread *pProc);
    CProcessThread *    FindProcess(DWORD dwProcId);
    HRESULT             get_StatusValue(long nIndex, long *pnStatus);
    HRESULT             put_StatusValue(long nIndex, long nStatus);

    // Hack function to work around JSCRIPT.DLL bug
    HRESULT             HackCreateInstance(REFCLSID, IUnknown *, DWORD, REFIID, LPVOID*);

    BOOL SetScriptPath(const TCHAR *pszScriptPath, const TCHAR *pszInitScript);

    BOOL               _fHackVersionChecked;
    IClassFactory *    _pJScriptFactory;

    HRESULT RunScript(LPWSTR bstrPath, VARIANT *pvarParams);

    TCHAR   *GetIdentity();
    void    SetStatusDialogCaption();

protected:
    virtual BOOL  Init();
    virtual DWORD ThreadMain();

    void    InitScriptDebugger();
    void    DeInitScriptDebugger();
    BOOL    ConfigureUI();
    void    CleanupUI();
    HRESULT LoadTypeLibraries();

    void ShowMenu(int x, int y);
    void Reboot();
    void Restart();
    void OpenStatusDialog();
    void HandleThreadMessage();

    HRESULT UpdateOptionSettings(BOOL fSave);

    void CleanupOldProcesses();

private:
    TCHAR           _tchIdentity[MAX_IDENTITY_LENGTH];
    BOOL            _fInDestructor;
    BOOL            _fRestarting;
    HWND            _hwnd;
    CStatusDialog   *_pStatusDialog;

public:
    struct OPTIONSETTINGS : public CThreadLock
    {
        OPTIONSETTINGS();

        static void GetModulePath(CStr *pstr);

        void GetScriptPath(CStr *cstrPage); // internally does a LOCK_LOCALS
        void GetInitScript(CStr *cstr); // internally does a LOCK_LOCALS

        CStr  cstrScriptPath;
        CStr  cstrInitScript;
    };

    ITypeLib                * _pTypeLibEXE;
    ITypeInfo               * _pTIMachine;
    IGlobalInterfaceTable * _pGIT;

    // _rgnStatusValues: Simple array of status values -- Multithreaded access, but no locking necessary
    long                     _rgnStatusValues[MAX_STATUS_VALUES];

    // **************************
    //   THREAD-SAFE MEMBER DATA
    //   All access to the following members must be protected by LOCK_LOCALS()
    //   or InterlockedXXX.
    //
    OPTIONSETTINGS  _options;
    CMachine*       _pMachine;

    VARIANT         _vPublicData;
    VARIANT         _vPrivateData;

    DWORD           _dwPublicDataCookie;
    DWORD           _dwPrivateDataCookie;

    DWORD           _dwPublicSerialNum;
    DWORD           _dwPrivateSerialNum;

    CStackPtrAry<CScriptHost*, 10> _aryScripts;

    CStackPtrAry<CProcessThread*, 10> _aryProcesses;
};

//+-----------------------------------------------------------------------
//
//  Class:      CMachine (cm)
```

```
//
//  Purpose:    Contains all useful info about a machine and what it's
//              doing.
//
//  Notes:      This class is manipulated from multiple threads. All
//              member functions must be thread safe!
//
//              This is the class that is created by the class factory and
//              handed out as a remote object to other machines. It has no
//              real code in itself but merely provides a way to talk to the
//              already running script engines.
//
//----------------------------------------------------------------------------

class CMachine : public CThreadComm,
                 public IConnectedMachine,
                 public IConnectionPointContainer
{
    friend class CMachConnectPoint;

public:
    DECLARE_MEMCLEAR_NEW_DELETE();

    CMachine(CMTScript *pMT, ITypeInfo *pTIMachine);
   ~CMachine();

    DECLARE_STANDARD_IUNKNOWN(CMachine);

    // IDispatch interface

    STDMETHOD(GetTypeInfoCount)(UINT FAR* pctinfo);

    STDMETHOD(GetTypeInfo)(
      UINT itinfo,
      LCID lcid,
      ITypeInfo FAR* FAR* pptinfo);

    STDMETHOD(GetIDsOfNames)(
  REFIID riid,
  OLECHAR FAR* FAR* rgszNames,
  UINT cNames,
  LCID lcid,
  DISPID FAR* rgdispid);

STDMETHOD(Invoke)(
  DISPID dispidMember,
  REFIID riid,
  LCID lcid,
  WORD wFlags,
  DISPPARAMS FAR* pdispparams,
  VARIANT FAR* pvarResult,
  EXCEPINFO FAR* pexcepinfo,
  UINT FAR* puArgErr);

// IConnectionPointContainer methods

STDMETHOD(EnumConnectionPoints)(LPENUMCONNECTIONPOINTS*);
STDMETHOD(FindConnectionPoint)(REFIID, LPCONNECTIONPOINT*);

// IConnectedMachine interface

STDMETHOD(Exec)(BSTR bstrCmd, BSTR bstrParams, VARIANT *pvData);

STDMETHOD(get_PublicData)(VARIANT *pvData);
STDMETHOD(get_Name)(BSTR *pbstrName);
STDMETHOD(get_Platform)(BSTR *pbstrPlat);
STDMETHOD(get_OS)(BSTR *pbstrOS);
STDMETHOD(get_MajorVer)(long *plMajorVer);
STDMETHOD(get_MinorVer)(long *plMinorVer);
STDMETHOD(get_BuildNum)(long *plBuildNum);
STDMETHOD(get_PlatformIsNT)(VARIANT_BOOL *pfIsNT);
STDMETHOD(get_ServicePack)(BSTR *pbstrSP);
STDMETHOD(get_HostMajorVer)(long *plMajorVer);
STDMETHOD(get_HostMinorVer)(long *plMajorVer);
STDMETHOD(get_StatusValue)(long nIndex, long *pnStatus);
```

```
    HRESULT FireScriptNotify(BSTR bstrIdent, VARIANT vInfoF);

    STDMETHOD(CreateIScriptedProcess)(long lProcessID, wchar_t *pszEnvID, IScriptedProcess **pISP);

    TCHAR* GetIdentity();
    void InitIdentity();

    #define LOCK_MACH_LOCALS(pObj)  CMachLock local_lock(pObj);

protected:
    virtual BOOL  Init();
    virtual DWORD ThreadMain();

    BOOL HandleThreadMessage();

private:
    class CMachLock
    {
    public:
        CMachLock(CMachine *pThis);
       ~CMachLock();

    private:
        CMachine *_pThis;
    };
    friend class CMachLock;

    CMTScript *                 _pMT;
    ITypeInfo *                 _pTypeInfoIMachine;
    TCHAR                       _tchIdentity[MAX_IDENTITY_LENGTH];


    CRITICAL_SECTION            _cs;
    CStackPtrAry<IDispatch*, 5> _aryDispSink;

};

inline
```

```
CMachine::CMachLock::CMachLock(CMachine *pThis)
    : _pThis(pThis)
{
    EnterCriticalSection(&_pThis->_cs);
}

inline
CMachine::CMachLock::~CMachLock()
{
    LeaveCriticalSection(&_pThis->_cs);
}

//+---------------------------------------------------------------------------
//
//  Class:      CProcessThread (cpt)
//
//  Purpose:    Class which spawns a process, monitors its success, talks to
//              it during execution if necessary, and returns its completion
//              status. (each CProcessThread is in its own thread)
//
//----------------------------------------------------------------------------

class CProcessThread  : public CThreadComm
{
public:
    CProcessThread(CScriptHost *pSH);
   ~CProcessThread();

    DECLARE_STANDARD_IUNKNOWN(CProcessThread);

    DWORD  ProcId()    { return _piProc.dwProcessId; }

    // Thread-Safe member functions. These can be called by any thread to
    // get the appropriate information without having to go through
    // PostToThread. These are only safe AFTER the process has been started.

    HRESULT GetProcessOutput(BSTR *pbstrOutput);
    DWORD   GetExitCode();
    void      SetExitCode(DWORD dwExitCode)
                  {
                      _dwExitCode = dwExitCode;
                      _fUseExitCode = TRUE;
                  }
    void      Terminate();

    ULONG     GetDeadTime();
    BOOL      IsOwner(DWORD dwProcID, long lID)
                    { return (lID == _lEnvID); }

    CScriptHost * ScriptHost()
                    { return _pSH; }

    void SetProcComm(CProcessComm *pPC)
                    { Assert(!_pPC || !pPC); _pPC = pPC; }
    CProcessComm * GetProcComm()
                    { return _pPC; }

    const PROCESS_PARAMS *GetParams() const { return &_ProcParams; }

    void SetIdentity(TCHAR* tchIdentity);
protected:

    virtual DWORD ThreadMain();
    virtual BOOL  Init();

    void HandleThreadMessage();
    void HandleProcessExit();
    BOOL IsDataInPipe();
    void ReadPipeData();
    void CheckIoPort();

    HRESULT LaunchProcess(const PROCESS_PARAMS *pProcParams);
    void    GetProcessEnvironment(CStr *pcstr, BOOL fNoEnviron);

private:
    TCHAR               _tchIdentity[MAX_IDENTITY_LENGTH];

    CScriptHost         *_pSH;
    CProcessComm        *_pPC;          // Not AddRef'd

    PROCESS_INFORMATION _piProc;
    long                _lEnvID;

    DWORD               _dwExitCode;    // Value set explicitely by the process
    BOOL                _fUseExitCode; // TRUE if _dwExitCode is the code we want

    HANDLE              _hPipe;
    BYTE                _abBuffer[PIPE_BUFFER_SIZE];

    HANDLE              _hJob;
    HANDLE              _hIoPort;

#if DBG == 1
    // These are used when tracking detailed information on child processes.
    CStackPtrAry<DWORD, 10>   _aryProcIds;
    CStackDataAry<HANDLE, 10> _aryProcHandles;
    CStackPtrAry<char*, 50>   _aryProcNames;
#endif

    CProcessParams      _ProcParams;

    _int64              _i64ExitTime;

    // Access to the following members must be thread-safe (by calling
    // LOCK_LOCALS).
    CStr                _cstrOutput;
};

//+---------------------------------------------------------------------------
//
//  Class:      CProcessComm (cpc)
//
//  Purpose:    Handles communication with another process on the local
//              machine via IScriptedProcess.
//
//---------------------------------------------------------------------------

class CProcessComm : public IScriptedProcess
{
public:

    CProcessComm(CMTScript *pMT);
   ~CProcessComm();

    DECLARE_MEMCLEAR_NEW_DELETE();

    DECLARE_STANDARD_IUNKNOWN(CProcessComm);

    // IScriptedProcess methods

    STDMETHOD(SetProcessID)(long lProcessID, wchar_t *pszEnvID);
    STDMETHOD(SendData)(wchar_t * pszType,
                        wchar_t * pszData,
                        long *plReturn);
    STDMETHOD(SetExitCode)(long lExitCode);
    STDMETHOD(SetProcessSink)(IScriptedProcessSink * pSPS);


    void SendToProcess(MACHPROC_EVENT_DATA *pmed);

private:

    CMTScript             *_pMT;
    IScriptedProcessSink *_pSink;
    CScriptHost           *_pSH;
    CProcessThread        *_pProc;
};

//+--------------------------------------------------------------------------
//
//  Class:      CScriptSite
```

```
//
//  Purpose:    Active scripting site
//
//---------------------------------------------------------------------------

class CScriptSite :
    public IActiveScriptSite,
    public IActiveScriptSiteWindow,
    public IActiveScriptSiteDebug,
    public IProvideMultipleClassInfo,
    public IConnectionPointContainer,
    public IGlobalMTScript
{
public:

     DECLARE_MEMCLEAR_NEW_DELETE();

     CScriptSite(CScriptHost * pSH);
    ~CScriptSite();

     HRESULT Init(LPWSTR pszName);
     void    Close();
     void    Abort();

     // IUnknown methods

     STDMETHOD_(ULONG, AddRef)();
     STDMETHOD_(ULONG, Release)();
     STDMETHOD(QueryInterface)(REFIID, void **);

     // IActiveScriptSite methods

     STDMETHOD(GetLCID)(LCID *plcid);
     STDMETHOD(GetItemInfo)(LPCOLESTR pstrName, DWORD dwReturnMask, IUnknown **ppiunkItem, ITypeInfo
**ppti);
     STDMETHOD(GetDocVersionString)(BSTR *pszVersion);
     STDMETHOD(RequestItems)(void);
     STDMETHOD(RequestTypeLibs)(void);
STDMETHOD(OnScriptTerminate)(const VARIANT *pvarResult, const EXCEPINFO *pexcepinfo);
STDMETHOD(OnStateChange)(SCRIPTSTATE ssScriptState);
STDMETHOD(OnScriptError)(IActiveScriptError *pscripterror);
STDMETHOD(OnEnterScript)(void);
STDMETHOD(OnLeaveScript)(void);

// IActiveScriptSiteWindow methods

STDMETHOD(GetWindow)(HWND *phwnd);
STDMETHOD(EnableModeless)(BOOL fEnable);

// IActiveScriptSiteDebug methods

STDMETHOD(GetDocumentContextFromPosition)(DWORD dwSourceContext,
                                          ULONG uCharacterOffset,
                                          ULONG uNumChars,
                                          IDebugDocumentContext **ppsc);

STDMETHOD(GetApplication)(IDebugApplication **ppda);
STDMETHOD(GetRootApplicationNode)(IDebugApplicationNode **ppdanRoot);
STDMETHOD(OnScriptErrorDebug)(IActiveScriptErrorDebug *pErrorDebug,
                              BOOL *pfEnterDebugger,
                              BOOL *pfCallOnScriptErrorWhenContinuing);

// IProvideClassInfo methods

STDMETHOD(GetClassInfo)(ITypeInfo **);
STDMETHOD(GetGUID)(DWORD dwGuidKind, GUID * pGUID);

// IProvideMultipleClassInfo methods

STDMETHOD(GetMultiTypeInfoCount)(ULONG *pcti);
STDMETHOD(GetInfoOfIndex)(ULONG iti, DWORD dwFlags, ITypeInfo** pptiCoClass, DWORD* pdwTIFlags, ULONG*
pcdispidReserved, IID* piidPrimary, IID* piidSource);

// IConnectionPointContainer methods

STDMETHOD(EnumConnectionPoints)(LPENUMCONNECTIONPOINTS*);
STDMETHOD(FindConnectionPoint)(REFIID, LPCONNECTIONPOINT*);

// IBServer methods
// We need to implement these on a separate identity from
// the main pad object in order to prevent ref count loops
// with the script engine.

STDMETHOD(GetTypeInfoCount)(UINT FAR* pctinfo);

STDMETHOD(GetTypeInfo)(
  UINT itinfo,
  LCID lcid,
  ITypeInfo FAR* FAR* pptinfo);

STDMETHOD(GetIDsOfNames)(
  REFIID riid,
  OLECHAR FAR* FAR* rgszNames,
  UINT cNames,
  LCID lcid,
  DISPID FAR* rgdispid);

STDMETHOD(Invoke)(
  DISPID dispidMember,
  REFIID riid,
  LCID lcid,
  WORD wFlags,
  DISPPARAMS FAR* pdispparams,
  VARIANT FAR* pvarResult,
  EXCEPINFO FAR* pexcepinfo,
  UINT FAR* puArgErr);

STDMETHOD(get_PublicData)(VARIANT *);
STDMETHOD(put_PublicData)(VARIANT);
STDMETHOD(get_PrivateData)(VARIANT *);
STDMETHOD(put_PrivateData)(VARIANT);
STDMETHOD(ExitProcess)();
STDMETHOD(Restart)();
STDMETHOD(get_LocalMachine)(BSTR *);
```

```
    STDMETHOD(get_Identity)(BSTR *);
    STDMETHOD(Include)(BSTR);
    STDMETHOD(CallScript)(BSTR, VARIANT *);
    STDMETHOD(SpawnScript)(BSTR, VARIANT *);
    STDMETHOD(get_ScriptParam)(VARIANT *);
    STDMETHOD(get_ScriptPath)(BSTR *);
    STDMETHOD(CallExternal)(BSTR, BSTR, VARIANT *, long *);
    STDMETHOD(ResetSync)(const BSTR);
    STDMETHOD(WaitForSync)(BSTR, long, VARIANT_BOOL *);
    STDMETHOD(WaitForMultipleSyncs)(const BSTR, VARIANT_BOOL, long, long *);
    STDMETHOD(SignalThreadSync)(BSTR);
    STDMETHOD(TakeThreadLock)(BSTR);
    STDMETHOD(ReleaseThreadLock)(BSTR);
    STDMETHOD(DoEvents)();
    STDMETHOD(MessageBoxTimeout)(BSTR, long, BSTR, long, long, VARIANT_BOOL, VARIANT_BOOL, long *);
    STDMETHOD(RunLocalCommand)(BSTR, BSTR, BSTR, VARIANT_BOOL, VARIANT_BOOL, VARIANT_BOOL, VARIANT_BOOL,
VARIANT_BOOL, long *);
    STDMETHOD(GetLastRunLocalError)(long *);
    STDMETHOD(GetProcessOutput)(long, BSTR *);
    STDMETHOD(GetProcessExitCode)(long, long *);
    STDMETHOD(TerminateProcess)(long);
    STDMETHOD(SendToProcess)(long, BSTR, BSTR, long *);
    STDMETHOD(SendMail)(BSTR, BSTR, BSTR, BSTR, BSTR, BSTR, BSTR, BSTR, long *);
    STDMETHOD(SendSMTPMail)(BSTR, BSTR, BSTR, BSTR, BSTR, BSTR, long *);
    STDMETHOD(ASSERT)(VARIANT_BOOL, BSTR);
    STDMETHOD(OUTPUTDEBUGSTRING)(BSTR);
    STDMETHOD(UnevalString)(BSTR, BSTR*);
    STDMETHOD(CopyOrAppendFile)(BSTR bstrSrc,BSTR bstrDst,long nSrcOffset,long nSrcLength,VARIANT_BOOL
fAppend,long *nSrcFilePosition);
    STDMETHOD(Sleep)(int);
    STDMETHOD(Reboot)();
    STDMETHOD(NotifyScript)(BSTR, VARIANT);
    STDMETHOD(RegisterEventSource)(IDispatch *pDisp, BSTR bstrProgID);
    STDMETHOD(UnregisterEventSource)(IDispatch *pDisp);
    STDMETHOD(get_HostMajorVer)(long *pVer);
    STDMETHOD(get_HostMinorVer)(long *pVer);
    STDMETHOD(get_StatusValue)(long nIndex, long *pnStatus);
    STDMETHOD(put_StatusValue)(long nIndex, long nStatus);
    // Other methods

    HRESULT ExecuteScriptStr(TCHAR * pchScript);
    HRESULT ExecuteScriptFile(TCHAR *pchPath);
    HRESULT SetScriptState(SCRIPTSTATE ss);

    CScriptHost * ScriptHost() { return _pSH; }

    // Member variables
    CStr                        _cstrName;
    ULONG                       _ulRefs;
    CScriptSite *               _pScriptSitePrev;
    IActiveScript *             _pScript;
    CScriptHost*                _pSH;
    TCHAR                       _achPath[MAX_PATH];
    VARIANT                     _varParam;
    IDispatch *                 _pDispSink;
    IDebugDocumentHelper      * _pDDH;  // Script Debugging helper
    DWORD                       _dwSourceContext;
    BOOL                        _fInDebugError;
    TCHAR                       _tchIdentity[MAX_IDENTITY_LENGTH];

private:
    BOOL                        _fInScriptError;
};

class AutoCriticalSection : public CRITICAL_SECTION
{
public:
    AutoCriticalSection()
    {
        InitializeCriticalSection(this);
    }
    ~AutoCriticalSection()
    {
        DeleteCriticalSection(this);
    }
```

```
};

class CScriptHost :
        public CThreadComm,
        public IGlobalMTScript
{
    friend class CScriptEventSink;

public:
    DECLARE_MEMCLEAR_NEW_DELETE();

    CScriptHost(CMTScript *  pBS,
                BOOL         fPrimary,
                BOOL         fDispatchOnly);

   ~CScriptHost();

    DECLARE_STANDARD_IUNKNOWN(CScriptHost);

    // Script management

    HRESULT LoadTypeLibrary();
    HRESULT PushScript(TCHAR *pchType);
    HRESULT PopScript();
    HRESULT CloseScripts();
    HRESULT AbortScripts();
    HRESULT ExecuteTopLevelScript(TCHAR *pchPath, VARIANT *pvarParams);
    HRESULT ExecuteTopLevelScriptlet(TCHAR *pchScript);

    long    FireScriptErrorEvent(
                              TCHAR *bstrFile,
                              long nLine,
                              long nChar,
                              TCHAR *bstrText,
                              long sCode,
                              TCHAR *bstrSource,
                              TCHAR *bstrDescription);
    long    FireScriptErrorEvent(TCHAR *szMsg);
```

```
void     FireProcessEvent(THREADMSG mt, CProcessThread *pProc);
void     FireMachineEvent(MACHPROC_EVENT_DATA *pmed, BOOL fExec);
void     FireEvent(DISPID, UINT cArg, VARIANTARG *pvararg, VARIANTARG *pvarResult);
void     FireEvent(DISPID, UINT cArg, VARIANTARG *pvararg);
void     FireEvent(DISPID, LPCTSTR);
void     FireEvent(DISPID, BOOL);
void     FireEvent(DISPID, IDispatch *pDisp);

BOOL     GetMachineDispatch(LPSTR achName,
                            IConnectedMachine **ppMach);

void     SetIdentity(TCHAR *tchIdentity);


static HRESULT GetSyncEventName(int nEvent, CStr *pCStr, HANDLE *phEvent);
static HRESULT GetSyncEvent(LPCTSTR pszName, HANDLE *phEvent);
// IDispatch interface

STDMETHOD(GetTypeInfoCount)(UINT FAR* pctinfo);

STDMETHOD(GetTypeInfo)(
  UINT itinfo,
  LCID lcid,
  ITypeInfo FAR* FAR* pptinfo);

STDMETHOD(GetIDsOfNames)(
  REFIID riid,
  OLECHAR FAR* FAR* rgszNames,
  UINT cNames,
  LCID lcid,
  DISPID FAR* rgdispid);

STDMETHOD(Invoke)(
  DISPID dispidMember,
  REFIID riid,
  LCID lcid,
  WORD wFlags,
  DISPPARAMS FAR* pdispparams,
    VARIANT FAR* pvarResult,
    EXCEPINFO FAR* pexcepinfo,
    UINT FAR* puArgErr);

  // IGlobalMTScript interface

  STDMETHOD(get_PublicData)(VARIANT *);
  STDMETHOD(put_PublicData)(VARIANT);
  STDMETHOD(get_PrivateData)(VARIANT *);
  STDMETHOD(put_PrivateData)(VARIANT);
  STDMETHOD(ExitProcess)();
  STDMETHOD(Restart)();
  STDMETHOD(get_LocalMachine)(BSTR *);
  STDMETHOD(get_Identity)(BSTR *);
  STDMETHOD(Include)(BSTR);
  STDMETHOD(CallScript)(BSTR, VARIANT *);
  STDMETHOD(SpawnScript)(BSTR, VARIANT *);
  STDMETHOD(get_ScriptParam)(VARIANT *);
  STDMETHOD(get_ScriptPath)(BSTR *);
  STDMETHOD(CallExternal)(BSTR, BSTR, VARIANT *, long *);
  STDMETHOD(ResetSync)(const BSTR);
  STDMETHOD(WaitForSync)(BSTR, long, VARIANT_BOOL *);
  STDMETHOD(WaitForMultipleSyncs)(const BSTR, VARIANT_BOOL, long, long *);
  STDMETHOD(SignalThreadSync)(BSTR);
  STDMETHOD(TakeThreadLock)(BSTR);
  STDMETHOD(ReleaseThreadLock)(BSTR);
  STDMETHOD(DoEvents)();
  STDMETHOD(MessageBoxTimeout)(BSTR, long, BSTR, long, long, VARIANT_BOOL, VARIANT_BOOL, long *);
  STDMETHOD(RunLocalCommand)(BSTR, BSTR, BSTR, VARIANT_BOOL, VARIANT_BOOL, VARIANT_BOOL, VARIANT_BOOL,
VARIANT_BOOL, long *);
  STDMETHOD(GetLastRunLocalError)(long *);
  STDMETHOD(GetProcessOutput)(long, BSTR *);
  STDMETHOD(GetProcessExitCode)(long, long *);
  STDMETHOD(TerminateProcess)(long);
  STDMETHOD(SendToProcess)(long, BSTR, BSTR, long *);
  STDMETHOD(SendMail)(BSTR, BSTR, BSTR, BSTR, BSTR, BSTR, BSTR, BSTR, long *);
  STDMETHOD(SendSMTPMail)(BSTR, BSTR, BSTR, BSTR, BSTR, BSTR, long *);
  STDMETHOD(ASSERT)(VARIANT_BOOL, BSTR);
```

```
    STDMETHOD(OUTPUTDEBUGSTRING)(BSTR);
    STDMETHOD(UnevalString)(BSTR, BSTR*);
    STDMETHOD(CopyOrAppendFile)(BSTR bstrSrc,BSTR bstrDst,long nSrcOffset,long nSrcLength,VARIANT_BOOL
fAppend,long *nSrcFilePosition);
    STDMETHOD(Sleep)(int);
    STDMETHOD(Reboot)();
    STDMETHOD(NotifyScript)(BSTR, VARIANT);
    STDMETHOD(RegisterEventSource)(IDispatch *pDisp, BSTR bstrProgID);
    STDMETHOD(UnregisterEventSource)(IDispatch *pDisp);
    STDMETHOD(get_HostMajorVer)(long *pVer);
    STDMETHOD(get_HostMinorVer)(long *pVer);
    STDMETHOD(get_StatusValue)(long nIndex, long *pnStatus);
    STDMETHOD(put_StatusValue)(long nIndex, long nStatus);

    CScriptSite *  GetSite() { return _pScriptSite; }
    void GetScriptPath(CStr *pcstrPath);

    CMTScript *                  _pMT;
    CScriptSite *                _pScriptSite;

    BOOL                         _fIsPrimaryScript;
    BOOL                         _fMustExitThread;
    BOOL                         _fDontHandleEvents;
    ITypeInfo *                  _pTypeInfoIGlobalMTScript;
    ITypeInfo *                  _pTypeInfoCMTScript;
    ITypeLib *                   _pTypeLibEXE;

    VARIANT                      _vPubCache;
    VARIANT                      _vPrivCache;
    DWORD                        _dwPublicSN;
    DWORD                        _dwPrivateSN;

    long                         _lTimerInterval;

    HRESULT                      _hrLastRunLocalError;

    TCHAR                        _tchIdentity[MAX_IDENTITY_LENGTH];
    CStackPtrAry<CScriptEventSink*, 5> _aryEvtSinks;

protected:
    virtual DWORD ThreadMain();
    void HandleThreadMessage();

    enum MEP_RETURN
    {
        MEP_TIMEOUT,       // Timeout period expired
        MEP_EXIT,          // Thread is terminating
        MEP_FALLTHROUGH,   // No event occurred (fWait==FALSE only)
        MEP_EVENT_0,       // The given event(s) are signaled
    };

    DWORD MessageEventPump(BOOL     fWait,
                           UINT     cEvents   = 0,
                           HANDLE * pEvents   = NULL,
                           BOOL     fAll      = FALSE,
                           DWORD    dwTimeout = INFINITE,
                           BOOL     fNoEvents = FALSE);


    HRESULT StringToEventArray(const wchar_t *pszNameList, CStackPtrAry<HANDLE, 5> *aryEvents);
    HRESULT GetLockCritSec(LPTSTR               pszName,
                           CRITICAL_SECTION **ppcs,
                           DWORD            **ppdwOwner);

    struct SYNCEVENT
    {
        CStr   _cstrName;
        HANDLE _hEvent;
    };

    struct THREADLOCK
    {
        CStr             _cstrName;
        CRITICAL_SECTION _csLock;
        DWORD            _dwOwner;
    };

    // MAX_LOCKS is used because you can't move critical section objects
    // in memory once you've initialized them, thus making it impossible to
    // use the dynamic array class.
    #define MAX_LOCKS 10

    // The primary thread owns initialization and cleanup of these objects.
    static CStackDataAry<SYNCEVENT, 5> s_arySyncEvents;
    static THREADLOCK                  s_aThreadLocks[MAX_LOCKS];
    static UINT                        s_cThreadLocks;
    static AutoCriticalSection         s_csSync;
};
```